



下载APP



## 01 | net/http : 使用标准库搭建Server并不是那么简单

2021-09-13 叶剑峰

《手把手带你写一个Web框架》

课程介绍 >



讲述：叶剑峰

时长 18:25 大小 16.88M



你好，我是轩脉刃。欢迎加入我的课程，和我一起从 0 开始构建 Web 框架。

之前我简单介绍了整个课程的设计思路，也是搭建 Web 框架的学习路径，我们会先基于标准库搭建起 Server，然后一步一步增加控制器、路由、中间件，最后完善封装和重启，在整章学完后，你就能建立起一套自己的 Web 框架了。

其实你熟悉 Golang 的话就会知道，用官方提供的 net/http 标准库搭建一个 Web Server，是一件非常简单的事。我在面试的时候也发现，不少同学，在怎么搭怎么用的问题上，回答的非常溜，但是再追问一句为什么这个 Server 这么设计，涉及的 net/http 原理是什么？一概不知。



这其实是非常危险的。**实际工作中，我们会因为不了解底层原理，想当然的认为它的使用方式**，直接导致在代码编写、应用调优的时候出现各种问题。

所以今天，我想带着你从最底层的 HTTP 协议开始，搞清楚 Web Server 本质，通过 net/http 代码库梳理 HTTP 服务的主流程脉络，先知其所以然，再搭建框架的 Server 结构。

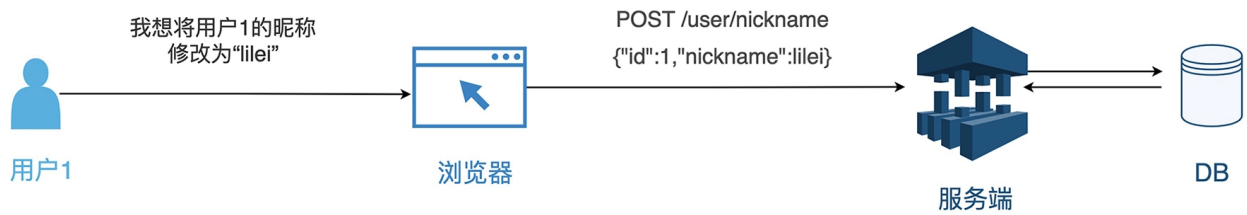
之后，我们会基于今天分析的整个 HTTP 服务主流程原理继续开发。所以这节课你掌握的程度，对于后续内容的理解至关重要。

## Web Server 的本质

既然要搭 Web Server，那我也先简单介绍一下，维基百科上是这么解释的，Web Server 是一个通过 HTTP 协议处理 Web 请求的计算机系统。这句话乍听有点绕口，我给你解释下。

HTTP 协议，在 OSI 网络体系结构中，是基于 TCP/IP 之上第七层应用层的协议，全称叫做超文本传输协议。啥意思？就是说 HTTP 协议传输的都是文本字符，只是这些字符是有规则排列的。这些字符的排列规则，就是一种约定，也就是协议。这个协议还有一个专门的描述文档，就是 [RFC 2616](#)。

对于 HTTP 协议，无论是请求还是响应，传输的消息体都可以分为两个部分：HTTP 头部和 HTTP Body 体。头部描述的一般是和业务无关但与传输相关的信息，比如请求地址、编码格式、缓存时长等；Body 里面主要描述的是与业务相关的信息。



Web Server 的本质，实际上就是**接收、解析** HTTP 请求传输的文本字符，理解这些文本字符的指令，然后进行**计算**，再将返回值**组织成** HTTP 响应的文本字符，通过 TCP 网络**传输回去**。

理解了 Web Server 干的事情，我们接下来继续看看在语言层面怎么实现。

## 一定要用标准库吗

对 Web Server 来说，Golang 提供了 net 库和 net/http 库，分别对应 OSI 的 TCP 层和 HTTP 层，它们两个负责的就是 HTTP 的接收和解析。

一般我们会使用 net/http 库解析 HTTP 消息体。但是可能会有人问，如果我想实现 Web 服务，可不可以不用 net/http 库呢？比如我直接用 net 库，逐字读取消息体，然后自己解析获取的传输字符。

答案是可以的，如果你有兼容其它协议、追求极致性能的需求，而且你有把握能按照 HTTP 的 RFC 标准进行解析，那完全可以自己封装一个 HTTP 库。

其实在一些大厂中确实是这么做的，每当有一些通用的协议需求，比如一个服务既要支持 HTTP，又要支持 Protocol Buffers，又或者想要支持自定义的协议，那么他们就可能抛弃 HTTP 库，甚至抛弃 net 库，直接自己进行网络事件驱动，解析 HTTP 协议。

有个开源库，叫 [FastHTTP](#)，它就是抛弃标准库 net/http 来实现的。作者为了追求极高的 HTTP 性能，自己封装了网络事件驱动，解析了 HTTP 协议。你感兴趣的话，可以去看看。

但是现在绝大部分的 Web 框架，都是基于 net/http 标准库的。我认为原因主要有两点：

第一是**相信官方开源的力量**。自己实现 HTTP 协议的解析，不一定会比标准库实现得更好，即使当前标准库有一些不足之处，我们也都相信，随着开源贡献者越来越多，标准库也会最终达到完美。

第二是 **Web 服务架构的变化**。随着容器化、Kubernetes 等技术的兴起，业界逐渐达成共识，单机并发性能并不是评判 Web 服务优劣的唯一标准了，易用性、扩展性也是底层库需要考量的。

所以总体来说，net/http 标准库，作为官方开源库，其易用性和扩展性都经过开源社区和 Golang 官方的认证，是我们目前构建 Web Server 首选的 HTTP 协议库。

用 net/http 来创建一个 HTTP 服务，其实很简单，下面是 [官方文档](#) 里的例子。我做了些注释，帮你理解。

[复制代码](#)

```
1 // 创建一个Foo路由和处理函数
2 http.Handle("/foo", fooHandler)
3
4 // 创建一个bar路由和处理函数
5 http.HandleFunc("/bar", func(w http.ResponseWriter, r *http.Request) {
6     fmt.Fprintf(w, "Hello, %q", html.EscapeString(r.URL.Path))
7 })
8
9 // 监听8080端口
10 log.Fatal(http.ListenAndServe(":8080", nil))
```

是不是代码足够简单？一共就 5 行，但往前继续推进之前，我想先问你几个问题，**这五行代码做了什么，为什么就能启动一个 HTTP 服务，具体的逻辑是什么样的？**

要回答这些问题，你就要深入理解 net/http 标准库。要不然，只会简单调用，却不知道原理，后面哪里出了问题，或者你想调优，就无从下手了。

所以，我们先来看看 net/http 标准库，从代码层面搞清楚整个 HTTP 服务的主流程原理，最后再基于原理讲实现。

## net/http 标准库怎么学

想要在 net/http 标准库纷繁复杂的代码层级和调用中，弄清楚主流程不是一件容易事。要快速熟悉一个标准库，就得找准方法。

**这里我教给你一个快速掌握代码库的技巧：库函数 > 结构定义 > 结构函数。**

简单来说，就是当你在阅读一个代码库的时候，不应该从上到下阅读整个代码文档，而应该先阅读整个代码库提供的对外库函数（function），再读这个库提供的结构（struct/class），最后再阅读每个结构函数（method）。



为什么要这么学呢？因为这种阅读思路和代码库作者的思路是一致的。

首先搞清楚这个库要提供什么功能（提供什么样的对外函数），然后为了提供这些功能，我要把整个库分为几个核心模块（结构），最后每个核心模块，我应该提供什么样的能力（具体的结构函数）来满足我的需求。

## 库函数（功能）

按照这个思路，我们来阅读 net/http 库，先看提供的对外库函数是为了实现哪些功能。这里顺带补充说明一下，我们课程对应的 Golang 源码的版本是 1.15.5，你可以在 [01 分支的 coredemo/go.mod](#) 里看到。

你直接通过 `go doc net/http | grep "^func"` 命令行能查询出 net/http 库所有的对外库函数：

[复制代码](#)

```
1 func CanonicalHeaderKey(s string) string
2 func DetectContentType(data []byte) string
3 func Error(w ResponseWriter, error string, code int)
4 func Get(url string) (resp *Response, err error)
5 func Handle(pattern string, handler Handler)
6 func HandleFunc(pattern string, handler func(ResponseWriter, *Request))
7 func Head(url string) (resp *Response, err error)
8 func ListenAndServe(addr string, handler Handler) error
9 func ListenAndServeTLS(addr, certFile, keyFile string, handler Handler) error
10 func MaxBytesReader(w ResponseWriter, r io.ReadCloser, n int64) io.ReadCloser
11 func NewRequest(method, url string, body io.Reader) (*Request, error)
12 func NewRequestWithContext(ctx context.Context, method, url string, body io.Re
13 func NotFound(w ResponseWriter, r *Request)
14 func ParseHTTPVersion(vers string) (major, minor int, ok bool)
15 func ParseTime(text string) (t time.Time, err error)
16 func Post(url, contentType string, body io.Reader) (resp *Response, err error)
17 func PostForm(url string, data url.Values) (resp *Response, err error)
18 func ProxyFromEnvironment(req *Request) (*url.URL, error)
19 func ProxyURL(fixedURL *url.URL) func(*Request) (*url.URL, error)
20 func ReadRequest(b *bufio.Reader) (*Request, error)
21 func ReadResponse(r *bufio.Reader, req *Request) (*Response, error)
22 func Redirect(w ResponseWriter, r *Request, url string, code int)
23 func Serve(l net.Listener, handler Handler) error
24 func ServeContent(w ResponseWriter, req *Request, name string, modtime time.Ti
25 func ServeFile(w ResponseWriter, r *Request, name string)
26 func ServeTLS(l net.Listener, handler Handler, certFile, keyFile string) error
27 func SetCookie(w ResponseWriter, cookie *Cookie)
28 func StatusText(code int) string
```

在这个库提供的方法中，我们去掉一些 New 和 Set 开头的函数，因为你从命名上可以看出，这些函数是对某个对象或者属性的设置。

剩下的函数大致可以分成三类：

为服务端提供创建 HTTP 服务的函数，名字中一般包含 Serve 字样，比如 Serve、ServeFile、ListenAndServe 等。

为客户端提供调用 HTTP 服务的类库，以 HTTP 的 method 同名，比如 Get、Post、Head 等。

提供中转代理的一些函数，比如 ProxyURL、ProxyFromEnvironment 等。

我们现在研究的是，如何创建一个 HTTP 服务，所以关注包含 Serve 字样的函数就可以了。

 复制代码

```
1 // 通过监听的URL地址和控制器函数来创建HTTP服务
2 func ListenAndServe(addr string, handler Handler) error{}
3 // 通过监听的URL地址和控制器函数来创建HTTPS服务
4 func ListenAndServeTLS(addr, certFile, keyFile string, handler Handler) error{
5 // 通过net.Listener结构和控制器函数来创建HTTP服务
6 func Serve(l net.Listener, handler Handler) error{}
7 // 通过net.Listener结构和控制器函数来创建HTTPS服务
8 func ServeTLS(l net.Listener, handler Handler, certFile, keyFile string) error
```

## 结构定义（模块）

然后，我们过一遍这个库提供的所有 struct，看看核心模块有哪些，同样使用 go doc:

 复制代码

```
1 go doc net/http | grep "^type"|grep struct
```

你可以看到整个库最核心的几个结构：

 复制代码

```
1 type Client struct{ ... }
2 type Cookie struct{ ... }
3 type ProtocolError struct{ ... }
4 type PushOptions struct{ ... }
5 type Request struct{ ... }
6 type Response struct{ ... }
7 type ServeMux struct{ ... }
8 type Server struct{ ... }
9 type Transport struct{ ... }
```

看结构的名称或者 go doc 查看结构说明文档，能逐渐了解它们的功能：

Client 负责构建 HTTP 客户端；

Server 负责构建 HTTP 服务端；

ServerMux 负责 HTTP 服务端路由；

Transport、Request、Response、Cookie 负责客户端和服务端传输对应的不同模块。

现在通过库方法（function）和结构体（struct），我们对整个库的结构和功能有大致印象了。整个库承担了两部分功能，一部分是构建 HTTP 客户端，一部分是构建 HTTP 服务端。

构建的 HTTP 服务端除了提供真实服务之外，也能提供代理中转服务，它们分别由 Client 和 Server 两个数据结构负责。除了这两个最重要的数据结构之外，HTTP 协议的每个部分，比如请求、返回、传输设置等都有具体的数据结构负责。

## 结构函数（能力）

下面从具体的需求出发，我们来阅读具体的结构函数（method）。

我们当前的需求是创建 HTTP 服务，开头我举了一个最简单的例子：

 复制代码

```
1 // 创建一个Foo路由和处理函数
2 http.Handle("/foo", fooHandler)
3
4 // 创建一个bar路由和处理函数
5 http.HandleFunc("/bar", func(w http.ResponseWriter, r *http.Request) {
6     fmt.Fprintf(w, "Hello, %q", html.EscapeString(r.URL.Path))
7 })
8
9 // 监听8080端口
10 log.Fatal(http.ListenAndServe(":8080", nil))
```

我们跟着 http.ListenAndServe 这个函数来理一下 net/http 创建服务的主流程逻辑。

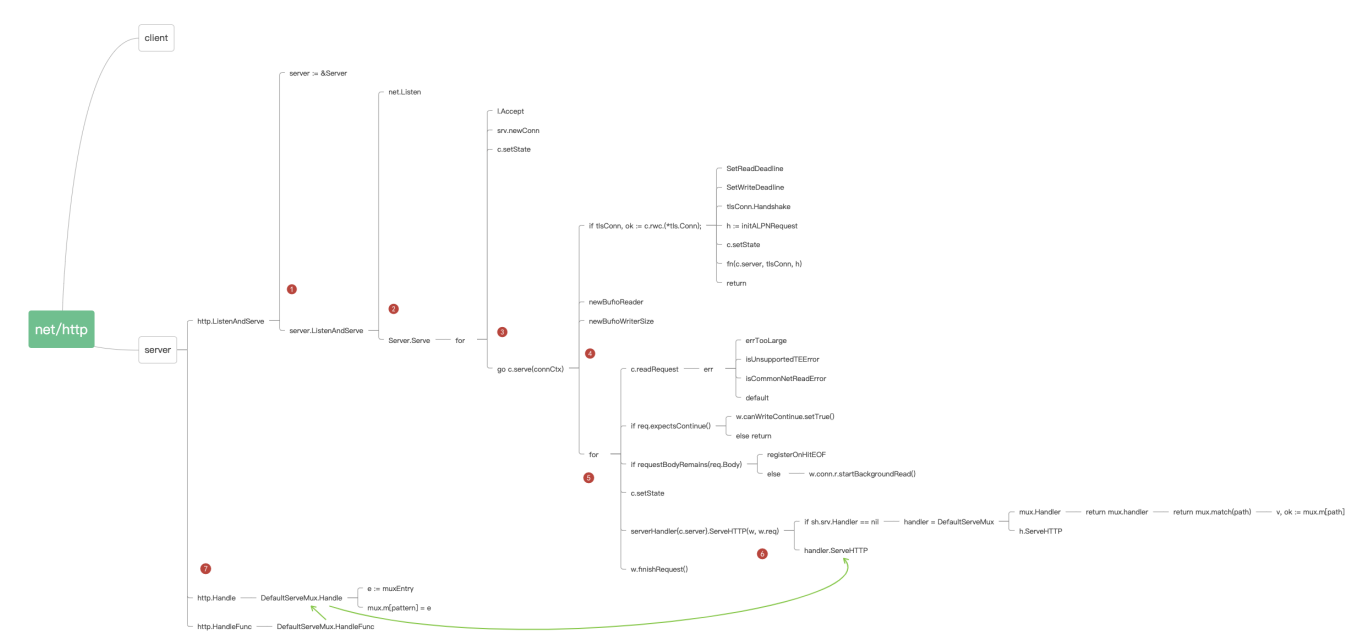
阅读具体的代码逻辑用 `go doc` 命令明显就不够了，你需要两个东西：

一个是可以灵活进行代码跳转的 IDE，VS Code 和 GoLand 都是非常好的工具。以我们现在要查看的 `http.ListenAndServe` 这个函数为例，我们可以从上面的例子代码中，直接通过 IDE 跳转到这个函数的源码中阅读，有一个能灵活跳转的 IDE 工具是非常必要的。

另一个是可以方便记录代码流程的笔记，这里我的个人方法是使用思维导图。

具体方法是**将要分析的代码从入口处一层层记录下来，每个函数，我们只记录其核心代码，然后对每个核心代码一层层解析**。记得把思维导图的结构设置为右侧分布，这样更直观。

比如下面这张图，就是我解析部分 HTTP 库服务端画的 **代码分析图**。

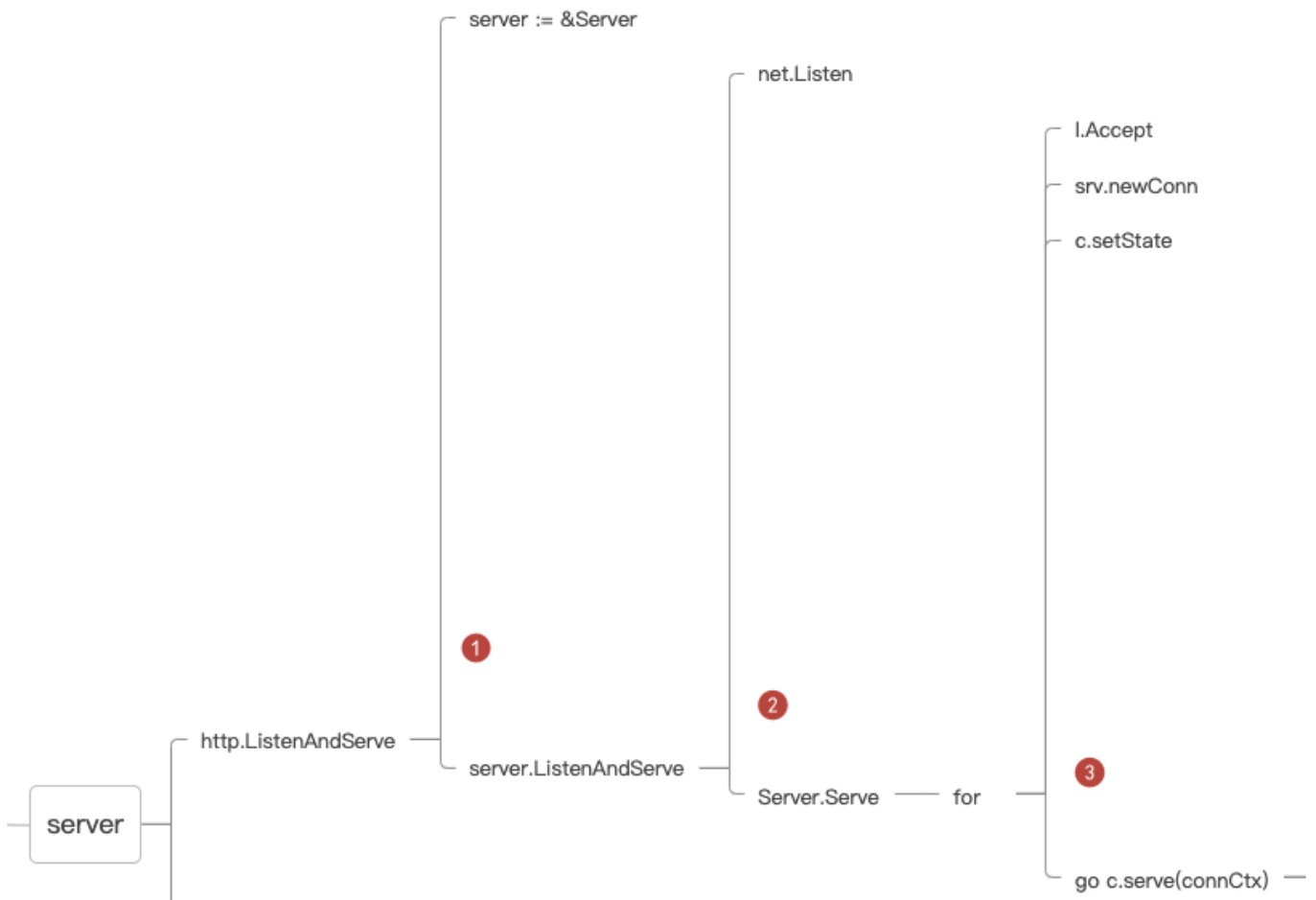


这张图看上去层级复杂，不过不用担心，对照着思维导图，我带你一层一层阅读，讲解每一层的逻辑，带你看清楚代码背后的设计思路。

我们先顺着 `http.ListenAndServe` 的脉络读。

**第一层**，`http.ListenAndServe` 本质是通过创建一个 `Server` 数据结构，调用 `server.ListenAndServe` 对外提供服务，这一层完全是比较简单的封装，目的是，将

Server 结构创建服务的方法 ListenAndServe , 直接作为库函数对外提供, 增加库的易用性。



进入到**第二层**, 创建服务的方法 ListenAndServe 先定义了监听信息 net.Listen, 然后调用 Serve 函数。

而在**第三层** Serve 函数中, 用了一个 for 循环, 通过 l.Accept 不断接收从客户端传进来的请求连接。当接收到了一个新的请求连接的时候, 通过 srv.NewConn 创建了一个连接结构 (http.conn), 并创建一个 Goroutine 为这个请求连接对应服务 (c.serve)。

从第四层开始, 后面就是单个连接的服务逻辑了。



在**第四层**，`c.serve`函数先判断本次 HTTP 请求是否需要升级为 HTTPS，接着创建读文本的 reader 和写文本的 buffer，再进一步读取本次请求数据，然后**第五层**调用最关键的方法 `serverHandler{c.server}.ServeHTTP(w, w.req)`，来处理这次请求。

这个关键方法是为了实现自定义的路由和业务逻辑，调用写法是比较有意思的：

```
1 serverHandler{c.server}.ServeHTTP(w, w.req)
```

复制代码

`serverHandler` 结构体，是标准库封装的，代表“请求对应的处理逻辑”，它只包含了一个指向总入口服务 `server` 的指针。

这个结构将总入口的服务结构 `Server` 和每个连接的处理逻辑巧妙联系在一起了，你可以看接着的**第六层**逻辑：

 复制代码

```

1 // serverHandler 结构代表请求对应的处理逻辑
2 type serverHandler struct {
3     srv *Server
4 }
5
6 // 具体处理逻辑的处理函数
7 func (sh serverHandler) ServeHTTP(rw ResponseWriter, req *Request) {
8     handler := sh.srv.Handler
9     if handler == nil {
10         handler = DefaultServeMux
11     }
12     ...
13     handler.ServeHTTP(rw, req)
14 }

```

如果入口服务 `server` 结构已经设置了 `Handler`，就调用这个 `Handler` 来处理此次请求，反之则使用库自带的 `DefaultServerMux`。

这里的 `serverHandler` 设计，能同时保证这个库的扩展性和易用性：你可以很方便使用默认方法处理请求，但是一旦有需求，也能自己扩展出方法处理请求。

那么 `DefaultServeMux` 是怎么寻找 `Handler` 的呢，这就是思维导图的最后一部分**第七层**。



`DefaultServeMux.Handle` 是一个非常简单的 `map` 实现，`key` 是路径（`pattern`），`value` 是这个 `pattern` 对应的处理函数（`handler`）。它是通过 `mux.match(path)` 寻找对应 `Handler`，也就是从 `DefaultServeMux` 内部的 `map` 中直接根据 `key` 寻找到 `value` 的。

这种根据 `map` 直接查找路由的方式是不是可以满足我们的路由需求呢？我们会在第三讲路由中详细解说。

好，HTTP 库 Server 的代码流程我们就梳理完成了，整个逻辑线大致是：

 复制代码

1 创建服务 -> 创建连接 -> 监听请求 -> 处理请求

如果你觉得层次比较多，对照着思维导图多看几遍就顺畅了。这里我也给你整理了一下逻辑线各层的关键结论：

第一层，标准库创建 HTTP 服务是通过创建一个 Server 数据结构完成的；

第二层，Server 数据结构在 for 循环中不断监听每一个连接；

第三层，每个连接默认开启一个 Goroutine 为其服务；

第四、五层，serverHandler 结构代表请求对应的处理逻辑，并且通过这个结构进行具体业务逻辑处理；

第六层，Server 数据结构如果没有设置处理函数 Handler，默认使用 DefaultServerMux 处理请求；

第七层，DefaultServerMux 是使用 map 结构来存储和查找路由规则。

如果你对上面几点关键结论还有疑惑的，可以再去看一遍思维导图。阅读核心逻辑代码是会有点枯燥，但是**这条逻辑线是 HTTP 服务启动最核心的主流程逻辑**，后面我们会基于这个流程继续开发，你要掌握到能背下来的程度。千万不要觉得要背诵了，压力太大，其实对照着思维导图，顺几遍逻辑，理解了再记忆就很容易。

## 创建框架的 Server 结构

现在原理弄清楚了，该下手搭 HTTP 服务了。

刚刚咱也分析了主流程代码，其中第一层的关键结论就是：net/http 标准库创建服务，实质上就是通过创建 Server 数据结构来完成的。所以接下来，我们就来创建一个 Server 数据结构。

通过 go doc net/http.Server 我们可以看到 Server 的结构：

```
1 type Server struct {  
2     // 请求监听地址  
3     Addr string  
4     // 请求核心处理函数  
5     Handler Handler  
6     ...  
7 }
```

[复制代码](#)

其中最核心的是 Handler 这个字段，从主流程中我们知道（第六层关键结论），当 Handler 这个字段设置为空的时候，它会默认使用 DefaultServerMux 这个路由器来填充这个值，但是我们一般都会使用自己定义的路由来替换这个默认路由。

所以在框架代码中，我们要创建一个自己的核心路由结构，实现 Handler。

先来理一下目录结构，我们在 [GitHub](#) 上创建一个项目 coredemo，这个项目是这门课程所有的代码集合，包含要实现的框架和使用框架的示例业务代码。

**所有的框架代码都存放在 framework 文件夹中，而所有的示例业务代码都存放在 framework 文件夹之外。**这里为了后面称呼方便，我们就把 framework 文件夹叫框架文件夹，而把外层称为业务文件夹。

当然 GitHub 上的这个 coredemo 是我在写课程的时候为了演示创建的，推荐你跟着一步一步写。成品在 [hade 项目](#) 里，你可以先看看，在最后发布的时候，我们会将整个项目进行发布。在一个新的业务中，如果要使用到我们自己写好的框架，可以直接通过引用 “import 项目地址 /framework” 来引入，在最后一部分做实战项目的时候我们会具体演示。

好，下面我们来一步步实现这个项目。

创建一个 framework 文件夹，新建 core.go，在里面写入。

```
1 package framework  
2  
3 import "net/http"  
4  
5 // 框架核心结构  
6 type Core struct {
```

[复制代码](#)

```
7 }
8
9 // 初始化框架核心结构
10 func NewCore() *Core {
11     return &Core{}
12 }
13
14 // 框架核心结构实现Handler接口
15 func (c *Core) ServeHTTP(response http.ResponseWriter, request *http.Request)
16     // TODO
17 }
18
```

而在业务文件夹中创建 main.go，其中的 main 函数就变成这样：

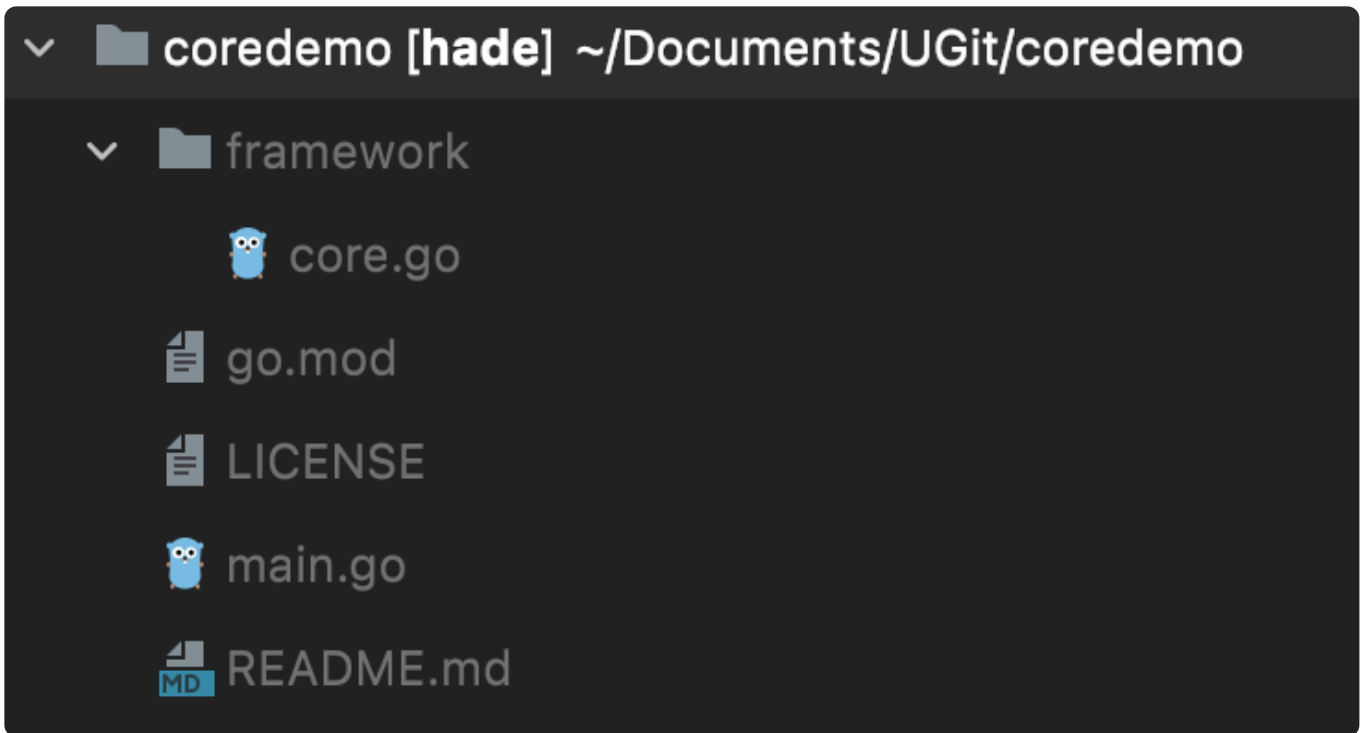
```
1 func main() {
2     server := &http.Server{
3         // 自定义的请求核心处理函数
4         Handler: framework.NewCore(),
5         // 请求监听地址
6         Addr:    ":8080",
7     }
8     server.ListenAndServe()
9 }
10
```

[复制代码](#)

整理下这段代码，我们通过自己创建了 Server 数据结构，并且在数据结构中创建了自定义的 Handler（Core 数据结构）和监听地址，实现了一个 HTTP 服务。这个服务的具体业务逻辑都集中在我们自定义的 Core 结构中，后续我们要做的事情就是不断丰富这个 Core 数据结构的功能逻辑。

后续每节课学完之后，我都会把代码放在对应的 GitHub 的分支中。你跟着课程敲完代码过程中有不了解的地方，可以对比参考分支。

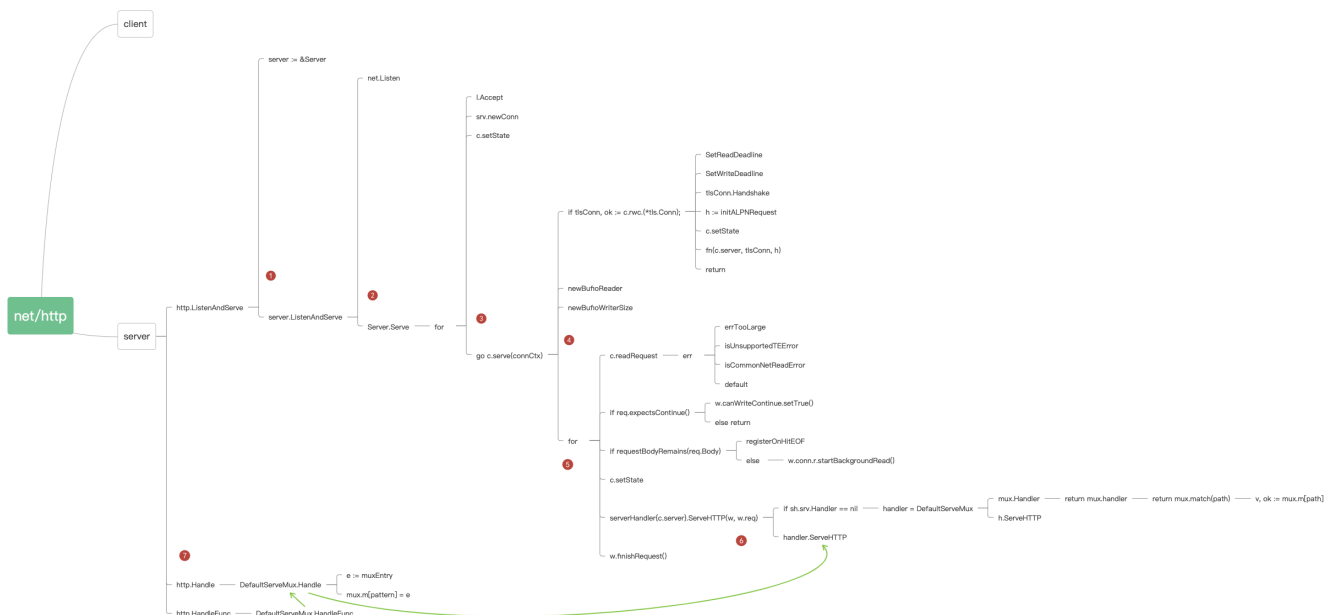
本节课我们完成的代码分支是：[@geekbang/01](#)，代码结构我也截了图：



## 小结

今天我以 net/http 标准库为例，分享了快速熟悉代码库的技巧，**库函数 > 结构定义 > 结构函数**。在阅读代码库时，从功能出发，先读对外库函数，再细读这个库提供的结构，搞清楚功能和对应结构之后，最后基于实际需求看每个结构函数。

读每个结构函数的时候，我们使用思维导图梳理了 net/http 创建 HTTP 服务的主流程逻辑，基于主流程原理，创建了一个属于我们框架的 Server 结构，你可以再回顾一下这张图。



主流程的链条比较长，但是你先理顺逻辑，记住几个关键的节点，再结合思维导图，就能记住整个主流程逻辑了，之后所有关于 HTTP 的细节和问题，我们都会基于这个主流程逻辑来思考和回答。

## 思考题

今天我用思维导图梳理了最核心的 HTTP 服务启动的主流程逻辑，知易行难，你不妨用这个思路做做下面这道思考题，尝试绘制出属于你自己的思维导图。

HTTP 库提供 FileServer 来封装对文件读取的 HTTP 服务。实现代码也非常简单：

 复制代码

```
1 fs := http.FileServer(http.Dir("/home/bob/static"))
2 http.Handle("/static/", http.StripPrefix("/static", fs))
```

请问它的主流程逻辑是什么？你认为其中最关键的节点是什么？

欢迎在留言区分享你的思考。感谢你的收听，如果你觉得今天的内容对你有帮助，也欢迎你把今天的内容分享给你身边的朋友，邀请他一起学习。

[点这里](#)进微信群，一起学习讨论。

分享给需要的人，Ta订阅后你可得 **20** 元现金奖励

 赞 10  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 [开篇词 | 为什么我们要从零开发框架？](#)

下一篇 [02 | Context：请求控制器，让每个请求都在掌控之中](#)

## 精选留言 (11)

 写留言

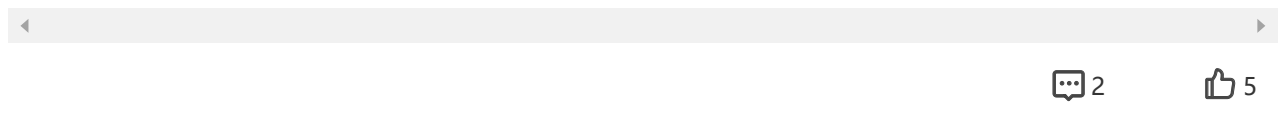
**Middleware**

2021-09-14

目录有点不清晰，从零开始，那么是不是应该给出建立合适的文件目录结构，命名。我们也能跟着上手敲一遍。比如这个 framework .目录是如何命名。希望老师真的能手把手

作者回复: 你好，在后面有一章会专门介绍目录的章节，叫《如何系统设计框架的整体目录》。每一个章节，我都有存放源码在github项目的对应分支，如果希望跟着上手敲一遍，可以跟着文章敲完代码，跟着对应分支对一遍。<https://github.com/gohade/coredemo/tree/geekbang/01>

谢谢支持，另外为你的ID middleware 点个赞。

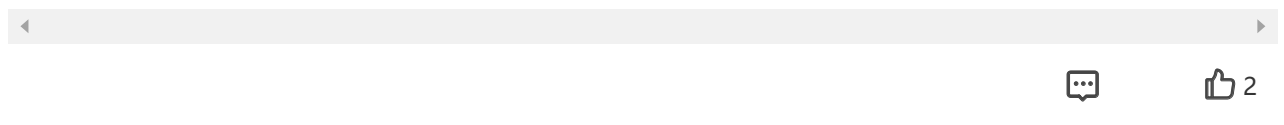
**return**

2021-09-14

很赞，读源码的思路和思维导图很值得学习

展开 ✓

作者回复: 恩，我平时阅读源码就是这样阅读的。经验之谈。感谢支持。

**进化菌**

2021-09-19

思维导图是个好东西，平时用 ide 点着一个个封装的函数、结构，跳着跳着脑子就乱了、懵了~

**C\_X**

2021-09-18

1. http.FileServer 创建 FileHandler 数据结构
  2. FileHandler 结构体中包含 FileSystem 接口，FileSystem 接口包含Open 方法
  3. http.Dir 的 Open 方法 实现 FileSystem 接口的 Open 方法
  4. http.Dir 的 Open 方法对表示字符串的文件路径进行判断：  
(1) 先判断 分隔符是否为 "/"且该字符串中是否包含分隔符，若不满足 返回 nil 和 e...
- 展开 ✓

**Groot**

2021-09-18

尝试画了一下脑图，感觉需要更加清楚原理，或者说知道每行代码都是在做什么事情之后，才能更好剥离出主逻辑代码

虽然还是没看懂源码，但按照这个方法隐约感觉有了一点进步，后续会继续按照这个方法多去看看源码...

展开 ∨

作者回复: 你好，脑图的方法阅读源码，我是一直用这种方式阅读的。非常有效果。



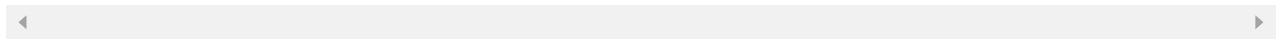
**Groot**

2021-09-18

一篇文章值回票价，感觉后续的文章都是在做慈善 🤖

受益匪浅，感谢分享 👍

作者回复: 你好，非常感谢支持。非常高兴本篇能让你有一些收获。后续文章是介绍写框架的过程的。感谢。



**布丁老厮**

2021-09-17

老师，HTTP 库 Server 的代码流程是不是应该为：创建服务 -> 监听请求 -> 创建连接 -> 处理请求 要更准确一点？因为net.Listen是在srv.newConn之前进行的。



**好家庭**

2021-09-16

老师，请问 go c.serve(connCtx) 里面为什么还有一个循环？c值得是一个connection，我理解不是每个连接处理一次就好了吗，为啥还有一个for循环呢？

...

/ Serve a new connection.

```
func (c *conn) serve(ctx context.Context) {...
```

展开 ∨



**逗逼章鱼**



2021-09-15

FileServer 的主要流程前面五层应该都一样，第六层开始不一样，FileServer --> fileHandler --> fileHandler里实现的 ServeHTTP --> serveFile 。

展开 ▾

**Ethan Liu**

2021-09-14

老师，思维导图用什么工具？

展开 ▾

作者回复: 用xmind或者wps

**bearlu**

2021-09-14

老师，这个课程源码分析的源码，基于哪个版本的golang？

作者回复: 你好，课程是基于1.15版本的golang进行分析的，可以看github上的go.mod中go的版本号

